Towards data-flow oriented work flow systems

Alexander Mattes[1], Annette Bieniusa[1], and Arie Middelkoop[2]

[1] University of Kaiserslautern
[2] vwd Group

Abstract. Work flow frameworks have become an integral part of modern information systems. They provide a clearly defined interface and structure for interaction with a system. The specification of work flow systems, however, is usually ad-hoc. Often, programmers simply define a number of tasks (forms and actions) that are sequentially connected. Users are then restricted to input data in this prescribed order.
In this paper, we propose a data-flow oriented work flow system where the data flow is described by a purely functional program. This approach offers the user and the system flexibility in the order of tasks while guaranteeing a consistent and correct result.

1 Motivation

Work flow frameworks are a common component in today's content management systems. They enable the modeling, structuring, support, and execution of business processes. In this paper, we focus on a particular type of interactive work flow, called case management [1], that dictates the tasks a user needs to perform with the system (e.g. fill in a form, print a document, acknowledge some information) to complete some business process. This could range from surveys or tax forms to complexly structured financial consultation sessions.

This particular type of work flow is typically described with control flow graphs (e.g. Apache ODE). The programming of such work flows and their tasks fits neatly the imperative paradigm. However, such an approach forces the user to perform the tasks in a rigid predefined order, even when this is not strictly necessary to stay in compliance with the company's policies and local laws.

For example, in financial or insurance consultation, the sessions are desirably dynamic, with the consultant jumping back and forth through tasks, possibly revising prior inputs, depending on the interaction with the client. Such behavior is difficult to describe cleanly with control flow graphs, especially when it contains dynamic elements like loops.

Therefore, we propose a functional approach, in which the work flows are described with data flow graphs instead. The nodes of this graphs represent tasks (effectively idempotent impure functions with localized side effects) and the edges represent pure functions that map outputs of tasks to inputs of (other) tasks. Arrows [4] provide the necessary scaffolding for the (dynamic) construction of such graphs.

Two extensions of arrows are particularly important in this context: *monadic arrows* allow the structure of the work flow to be dependent on inputs (e.g. to generate a set of tasks for each client), and *feedback loops* model destructive changes (e.g. finalizing or closing some tasks).

We present these ideas in a simplified setting as an embedded domain specific language using Haskell. We start with an example that demonstrates the notation, then discuss its properties and its implementation in more detail.

2 Example

As an example, we consider a simple work flow consisting of three individual tasks, given by the data flow chart in Figure 1. The tasks `askName` and `askAge` ask for the name and age of the user, respectively. The task `askAddress` asks for the address of the user if he is of full age, otherwise it asks for the address of a parent or legal guardian. Therefore it depends on the return value of `askAge`.

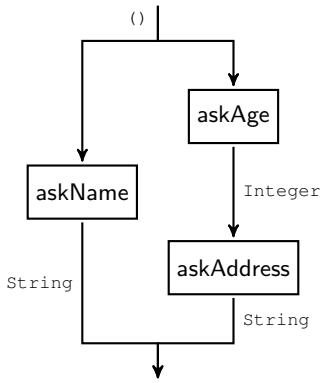

Fig. 1. Data flow of the example workflow.

While the data flow is fixed, the order in which the tasks have to be processed is only partially determined by the data dependencies. The only restriction is that the `askAge` task is executed before the `askAddress` task. Instead of programming the control flow by hand, we want to define the individual tasks independently of each other using pure and impure Haskell functions:

```
askName :: Task () String
askName = ...

askAge :: Task () Integer
askAge = ...

askAddress :: Task Integer String
askAddress = ...
```

In a second step, we combine them using `Arrow` constructors to model the data flow given by the chart:

```
workflow :: Task () (String,String)
workflow = askName &&& (askAge >>> askAddress)
```

Afterwards, it should be possible to execute the complete work flow using an automatically computed dynamic ordering of the tasks.

```
> runTask workflow ()
...
("Alice","Berlin")
```

where

```
runTask :: Task a b -> a -> IO b
runTask = ...
```

3 Implementation

In this section, we discuss the details of the task data type, the arrow combinators and execution of (composed) tasks in our Haskell implementation in detail.

3.1 Type Definition

We define a new parametric data type `Task a b` to model tasks with input type `a` and output type `b`. Since this type has to represent pure and impure tasks and support the various ways in which arrows can be combined, we need multiple constructors. In addition, we use a *generalized algebraic data type*(GADT) to achieve the necessary flexibility when composing tasks.

```
data Task a b where
    Pure     :: (a -> b) -> Task a b
    Impure   :: (a -> IO b) -> Task a b
    Serial   :: Task a b -> Task b c -> Task a c
    Parallel :: Task a b -> Task c d -> Task (a,c) (b,d)
```

The intended usage for the constructors is as follows:

`Pure` and `Impure` create single tasks out of pure functions and `IO` actions, respectively. While the `Impure` constructor would be sufficient for all functions (by simply lifting pure functions into the IO monad), this distinction allows us to later optimize the execution of the task; since pure functions have no side effects, the execution order is not important. In fact, we can rely on lazy evaluation to only compute those tasks whose results are actually needed.

`Serial` allows to compose two tasks in series, using the output of the first task as the input for the second. This dependency has to be considered later in the evaluation of the task. The `Serial` constructor basically models the composition `(>>>)` of arrows.

`Parallel` represents two tasks which are independent of each other and can thus be evaluated in an arbitrary order. It also represents the composition of two arrows with the `(***)` function.

3.2 Instance Declarations

The next step is to write an instance for the `Arrow` type class as well as for various subclasses. First, we have to turn `Task` into an instance of `Category`:

```
instance Cat.Category Task where
    id  = Pure id
    t1 . t2 = Serial t2 t1
```

The identity in `Catecory` is simply the pure task consisting of the identity function. The composition of two tasks is done with the `Serial` constructor.

Next, we can turn `Task` into an instance of the `Arrow` type class.

```
instance Arrow Task where
    arr       = Pure
    t1 *** t2 = Parallel t1 t2
    t1 &&& t2 = Pure (\a -> (a,a)) >>> t1 *** t2
    first t   = t *** Cat.id
    second t  = Cat.id *** t
```

The functions `arr` and `(***)` use the corresponding constructors `Pure` and `Parallel`. The `(&&&)` operator is implemented by using a pure function to feed the same input into both parallel tasks. `first` and `second` then simply use `(***)` and the identity task.

3.3 Additional Instances

To use the full potential of arrows, we can also create instances for the various arrow subclasses. For example, `ArrowChoice` allows the case distinction of two arrows based on the return value of the previous arrows and can be easily implemented by adding an additional constructor:

```
Or       :: Task a c -> Task b c -> Task (Either a b) c
```

The instantiation of `ArrowChoice` is based on using `Or` to represent the `(|||)` operator.

```
instance ArrowChoice Task where
  (|||)   = Or
  f +++ g = (f >>> arr Left) ||| (g >>> arr Right)
    left f  = f +++ Cat.id
    right f = Cat.id +++ f
```

Similarly, its possible to create instances for other subclasses like `ArrowApply` or `ArrowLoop`, if necessary, by adding further constructors to the data type.

3.4 Optimizing the Instances

The given instances can be optimized to simplify the resulting data structures. Using pattern matching, we can for example introduce special cases for pure tasks. The composition of two pure tasks can be done with the creation of a pure task containing the composition of the functions which results in a simpler data structure:

```
(Pure f1) . (Pure f2) = Pure (f1 . f2)
```

In the same way, the other arrow operators can be optimized for pure tasks by simply applying the operators to the contained functions and wrapping the result in a new pure task:

```
(Pure f1) *** (Pure f2) = Pure (f1 *** f2)
(Pure f1) &&& (Pure f2) = Pure (f1 &&& f2)
```

Notice that we cannot do the same with impure tasks since that would inevitably fix the execution order.

3.5 Arrow Laws

While `Task` is now technically an instance of the `Arrow` type class, we have to verify if it actually behaves like an arrow. As for many type classes, there are laws which every `Arrow` instance should obey [5]:

1. `arr id = id`
2. `arr (f >>> g) = arr f >>> arr g`
3. `first (arr f) = arr (first f)`
4. `first (f >>> g) = first f >>> first g`
5. `first f >>> arr fst = arr fst >>> f`
6. `first f >>> arr (id *** g) = arr (id *** g) >>> first f`
7. `first (first f) >>> arr assoc = arr assoc >>> first f`
   where `assoc ((a,b),c) = (a,(b,c))`

Note that there are similar laws for the `ArrowChoice` and `ArrowApply` type classes.

We want to check now if our implementation actually satisfies these laws. The first three are verified quite easily:

1.
$$\begin{aligned}\texttt{arr id} &= \texttt{Pure id}\\ &= \texttt{Category.id}\end{aligned}$$

2.
$$\begin{aligned}\texttt{arr (f >>> g)} &= \texttt{Pure (f >>> g)}\\ &= \texttt{Pure (g . f)}\\ &= \texttt{(Pure g) . (Pure f)}\\ &= \texttt{(arr g) . (arr f)}\\ &= \texttt{arr f >>> arr g}\end{aligned}$$

3.

```
first (arr f) = first (Pure f)
              = (Pure f) *** Cat.id
              = (Pure f) *** (Pure id)
              = (Pure f) *** (Pure id)
              = Pure (f *** id)
              = Pure (first f)
              = arr (first f)
```

For the fourth law, everything works as long as both tasks are pure:

4. (a) `f = Pure p` and `g = Pure q`:

```
first (f >>> g) = first (g . f)
                = first ((Pure q) . (Pure p))
                = first (Pure (q . p))
                = (Pure (q . p)) *** Cat.id
                = (Pure (q . p)) *** (Pure id)
                = Pure ((q . p) *** id)
                = Pure (first (q . p))
                = Pure (first (p >>> q))
                = Pure (first p >>> first q))
                = Pure ((p *** id) >>> (q *** id))
                = Pure (p *** id) >>> Pure (q *** id)
                = (Pure p) *** (Pure id) >>> (Pure q) *** (Pure id)
                = f *** Cat.id >>> g *** Cat.id
                = first f >>> first g
```

But if one of the tasks is not pure, the equality does not hold anymore:

(b) `f` $\neq$ `Pure p`:

```
first (f >>> g) = first (g . f)
                = first (Serial f g)
                = (Serial f g) *** Cat.id
                = Parallel (Serial f g) Cat.id
                ≠ Serial (f *** Cat.id) (Parallel g Cat.id)
                = Serial (f *** Cat.id) (g *** Cat.id)
                = Serial (first f) (first g)
                = first g . first f
                = first f >>> first g
```

The laws 5, 6 and 7 behave similarly. This means that only the first three laws hold in general; as soon as one of the tasks is not pure, they fail. This isn't surprising: For pure tasks, the laws are directly derivable from the fulfilled laws for the `Arrow` instance of functions. For all other tasks, the internal structure directly represents the order in which the arrows where combined. Hence, most laws have to fail. But is this a problem? The laws are there to guarantee that the instance actually behaves like an arrow. In our case, we are mostly interested in the behavior when we actually execute the task. It is therefore sufficient to check if the tasks are equivalent under execution.

3.6 Task execution

Until now, we considered the static representation of tasks as a composition of subtasks. Regarding their dynamic behavior, we needs a way to execute tasks. The `runTask` function executes a task as an IO action, thus allowing input and output of data:

```
runTask :: Task a b -> a -> IO b
```

The simplest way to do this is by remodeling the behavior of the *Kleisli arrow*.

```
runTask :: Task a b -> a -> IO b
runTask (Pure f)         = return . f
runTask (Impure m)       = m
runTask (Serial t1 t2)   = \a -> runTask t1 a >>= runTask t2
runTask (Parallel t1 t2) = \(u,v) -> do
                             r1 <- runTask t1 u
                             r2 <- runTask t2 v
                             return (r1,r2)
runTask (Or t1 t2)       = either (runTask t1) (runTask t2)
```

This certainly works, and it is also easy to show that this function is compatible with the arrow laws. For example, considering the fourth law,

```
first (f >>> g) = first f >>> first g
```

both sides of the equality evaluate to the execution of task `f` followed by the execution of task `g`.

While this solution respects the laws, it is not very useful yet. The execution order is fixed and not dynamic; in case of two parallel tasks, we always execute the left one first. To actually gain an advantage over simply using *Kleisli arrows*, we have to add some modifications. The easiest one would be to ask the user to specify the order in which parallel tasks get executed:

```
runTask (Parallel t1 t2) = \(u,v) -> do
                           putStrLn "Run Task a or b first?"
                           l <- getLine
                           if l == "b" then do
                               r2 <- (runTask t2) v
                               r1 <- (runTask t1) u
                               return (r1,r2)
                           else do
                               r1 <- (runTask t1) u
                               r2 <- (runTask t2) v
                               return (r1,r2)
```

This is an improvement but still not flexible enough. After choosing for example the left task, we have to finish all subtasks before we can start to execute the right task.

We can circumvent this problem by writing a function that executes only a single subtask:

```
runSingleTask :: (Task a b) -> a -> IO (Task a b)
runSingleTask t a = ...
```

We can then simply iterate over the subtasks of a composed task until all impure tasks have been processed:

```
runEveryTask :: (Task a b) -> a -> IO b
runEveryTask (Pure f) a = return . f $ a
runEveryTask t a = do
                s <- runSingleTask t a
                runEveryTask s a
```

To make this even more useful, we can add names

```
Impure   :: String -> (a -> IO b) -> Task a b
```

to the individual tasks to allow for an easier selection of which task to execute next.

A more advanced system could also offer the possibility to redo already processed tasks.

3.7 Example revisited

After all this work, we now return to our example from Section 2 and present an implementation. We start with the specification of the three in monadic syntax, assigning each a name:

```
askName :: Task () String
askName = Impure "askName" (const $ putStr "Please enter your name: " >>
     getLine)

askAge :: Task () Integer
askAge = Impure "askAge" (const $ putStr "Please enter your age: " >> readLn)

askAddress :: Task Integer String
askAddress = Impure "askAddress" $ \age -> do
        if age >= 18 then do
            putStr "Please enter your address: "
            getLine
        else do
            putStr "Please enter the address of a parent or legal guardian: "
            getLine
```

Then, we combine them with arrow operators reflecting the data dependencies to obtain the complete work flow:

```
workflow :: Task () (String,String)
workflow = askName &&& (askAge >>> askAddress)
```

When executing the tasks, we can dynamically choose the order in which the questions will be asked. For example, a session can take the following form:

```
> runEveryTask workflow ()
Open tasks:
0. askName
1. askAge
Which task do you want to run?: 1
Please enter your age: 23
Open tasks:
0. askName
1. askAddress
Which task do you want to run?: 0
Please enter your name: Alice
Please enter your address: Berlin
("Alice","Berlin")
```

4 Related work

The ideas in this paper are closely related to those of the iTasks system [6], which uses monads instead of arrows to describe work flows. Monads provide a means to describe a sequential work flow using pure functions. Recently, the authors seem to experiment with arrows as well [2]. Essential differences to our work are that we are not considering concurrent processes (orchestration), and instead enable different execution orders and revision of prior tasks.

These ideas are further a typical example of functional reactive programming [3], in particular with respect to the interaction of user and system.

5 Conclusion

In this paper, we presented a simple, but flexible work flow management framework. It allows to compose work flows from individual tasks and provides the means to process the subtasks in a dynamically adaptable order. A previous version of this work is incorporated in a real world financial application using a propriety functional programming language. Due to practical considerations, such as side effects in legacy code, that version used a mixture of control and data flow graphs. In this work, we essentially prototyped the next version which is based entirely on the work flow's data dependencies, and verified that the ideas are both viable and practical.

In future work, we will integrate our work flow management system into some web framework. This will provide programmers with the means to specify applications such as surveys and questionnaires in a simple and flexible way.

References

1. van der Aalst, W.M.P., Westergaard, M., Reijers, H.A.: Beautiful workflows: A matter of taste? In: Achten, P., Koopman, P.W.M. (eds.) The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday. Lecture Notes in Computer Science, vol. 8106, pp. 211–233. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40355-2

2. Achten, P., van Eekelen, M.C.J.D., de Mol, M., Plasmeijer, R.: Editorarrow: An arrow-based model for editor-based programming. J. Funct. Program. 23(2), 185–224 (2013)
3. Hudak, P.: Principles of functional reactive programming. ACM SIGSOFT Software Engineering Notes 25(1), 59 (2000)
4. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (May 2000), http://www.cs.chalmers.se/ rjmh/Papers/arrows.ps
5. Paterson, R.: "Control.Arrow". base-4.8.0.0: Basic libraries. haskell.org. Website (2002), retrieved 30 June 2015.
6. Plasmeijer, R., Achten, P., Koopman, P.W.M.: itasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007. pp. 141–152. ACM (2007), http://doi.acm.org/10.1145/1291151.1291174